Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland 20742

ABSTRACT

This paper discusses resource utilization over the life cycle of software development, and discusses the role that the current "waterfall model" plays in the actual software life cycle. The effects of prototyping are measured with respect to the life cycle model. Software production in the NASA environment was analyzed to measure these differences. The data collected from thirteen different projects and one prototype development were collected by the Software Engineering Laboratory at NASA Goddard Space Flight Center and analyzed for similarities and differences. The results indicate that the waterfall model is not very realistic in practice, and that a prototype development follows a similar life cycle as a production system--although, for this prototype, issues like system design and the user interface took precedence over issues such as correctness and robustness of the resulting system.

KEYWORDS: Life cycle, Measurement, Prototyping, Resource utilization, Waterfall chart

1. Introduction

As technology impacts the way industry builds software, there is increasing interest in understanding the software development model and in measuring both the process and product. New workstation technology, new languages (e.g., Ada, requirements and specification languages) as well as new techniques (e.g., prototyping, pseudocode) are impacting how software is built which further impacts how management needs to address these concerns in controlling and monitoring a software development.

In this paper, data are first presented which analyze several fairly large software projects from NASA Goddard Space Flight Center (GSFC) and put the current "waterfall" model in perspective. Data about software costs, productivity, reliability, modularity and other factors are collected by the Software Engineering Laboratory (SEL), a research group consisting of individuals from NASA/GSFC, Computer Sciences Corporation, and the University of Maryland, for research on improving both the software product and the process for building such software [SEL 82]. The Software Engineering Laboratory was established in 1976 to investigate the effectiveness of software engineering techniques for developing ground support software for NASA [BAS 78]. A recent prototyping experiment was conducted and data were collected which compare this prototype with the more traditional way to build software. The paper concludes with comments on the role of prototyping as a software development technique.

The software development process is typically product-driven, and can be divided into six major life cycle activities, each associated with a specific "end product" [WAS 83, ZEL 78]:

(1) *Requirements phase* and the publication of a requirements document.

(2) *Design phase* and the creation of a design document.

(3) *Code and Unit Test phase* and the generation of the source code library.

(4) *System integration and testing phase* and the fulfillment of the test plan.

(5) *Acceptance test phase* and completion of the acceptance test plan.

(6) *Operation and Maintenance phase* and the delivery of the completed system.

In order to present consistent data across a large number of projects, this paper only focuses on the interval between design and acceptance test and involves the actual implementation of the system by the developer group.

In this paper, we will refer to the term *activity* as the work required to complete a specific task. For example, the coding activity refers to all work done in generating the source code for a project, the design activity refers to building the program design, etc. On the other hand, the term *phase* will refer to that period of time when a certain activity is supposed to occur. For example, the Coding Phase will refer to that period of time during a software development when coding activities are supposed to occur. It is closely related to management-defined milestone dates for a project. But during this period, other activities may also occur.

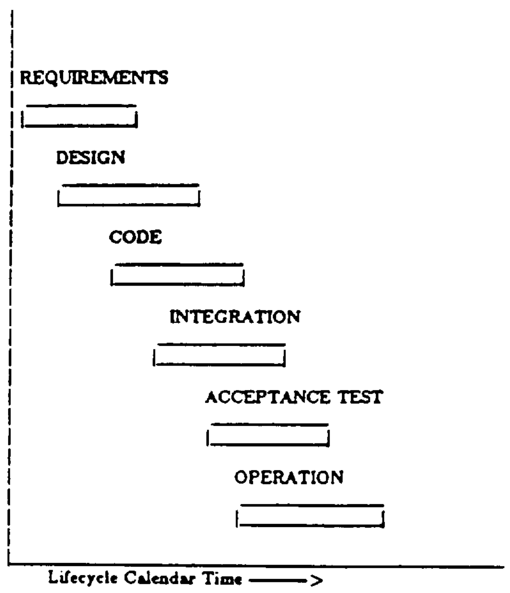

Figure 1. Typical Life Cycle

The waterfall model makes the assumption that all activity of a certain type occurs during the phase of that same name and phases do not overlap. Once a phase ends, then the next phase begins. Thus all requirements for a project occur during the Requirements Phase; all design activity occurs during the Design Phase. Once a project has a design review and enters the Coding Phase, then all activity is Coding. Since many companies keep data based upon hours worked by calendar date, this model is very easy to track. However, as Figure 1 shows, activities overlap and do not lie in separate phases. We will give more data on this later.

2. The waterfall chart is all wet

In the NASA/GSFC environment that we studied, the software life cycle follows a fairly standard set of activities [SEL 81]:

The *requirements* activity involves translating the functional specification consisting of physical attributes about the spacecraft to be launched into requirements for a software system that is to be built.

The *design* activity can be divided into two subactivities: the *preliminary design* activity and the *detailed design* activity. During preliminary design, the major subsystems are specified, input-output interfaces and implementation strategies are developed. During detailed design, the system architecture is extended to the subroutine and procedure level. Data structures and formal models of the system are defined. These models include procedural descriptions of the system, dataflow descriptions, complete description of all user input, system output, and input-output files, operational procedures, functional and procedural descriptions of each module, and complete description of all internal interfaces between modules.

The *Coding and Unit Test* activity involves the translation of the detailed design into a source program in some appropriate programming language (usually FORTRAN). Each programmer will unit test each module for apparent correctness.

The *System Integration and Test* activity validates that the completed system produced by the coding and unit test activity meets its specifications. Each module, as it is completed, is integrated into the growing system and integration test is performed to make sure that the entire package executes as expected. Functional testing of end-to-end system capabilities is performed according to the system test plan developed as part of the requirements activity.

In the *Acceptance Test* activity, the development team provides assistance to the acceptance test team, which checks that the system meets its requirements.

Operation and Maintenance activities begin after acceptance testing when the system becomes operational. For flight dynamics software at NASA, these activities are not significant to the overall cost. Most software produced is highly reliable. In addition, the flight dynamics software is usually not mission critical in that a failure of the software does not mean spacecraft failure but simply that the program has to be rerun. In addition, many of these programs (i.e., spacecraft) have limited lifetimes of six months to about three years.

Table 1 presents the raw data on the fourteen projects analyzed in this paper. The thirteen numbered projects are all fairly large flight dynamics programs, ranging in size from 15,500 lines of FORTRAN code to 89,513 lines of FORTRAN, with an average size of 57,890 lines of FORTRAN per system. The average work on these projects was 89.0 staff months; thus, all represent significant effort. The last project listed in Table 1 - FDAS - represents a prototype development and will be discussed in more detail later.

In most organizations, phase data are collected weekly so that they are the usual reporting mechanism. However, in the SEL, activity data are also collected. The data that are collected consist of nine possible activities for each component (i.e., source program module) worked on for that week. In this paper, these will be grouped as Design activities, Coding activities (code preparation and unit testing), Integration testing, Acceptance testing and Other. Specific review meetings, such as design reviews, will be grouped with their appropriate activity (e.g., a design review is a design activity, a code walkthrough is a coding activity, etc.). This allows us to look at both phase and activity utilization.

PROJECT SIZE AND STAFF-MONTH EFFORT

PROJECT NUMBER	SIZE (LINES OF CODE)	TOTAL EFFORT HOUR*	STAFF-MONTHS
1	15,500	17,715	116.5
2	50,911	12,588	82.8
3	61,178	17,039	112.1
4	26,844	10,946	72.0
5	25,731	1,514	10.0
6	67,325	19,475	128.4
7	66,260	17,997	118.4
8	+	+	+
9	55,237	15,262	100.4
10	75,420	5,792	38.1
11	89,513	15,122	99.5
12	75,393	14,508	95.4
13	85,369	14,309	94.1
Average	57,890	13,522	89.0
FDAS	33,967	14,150	93.1

+ - Raw data not available in data base
* - All technical effort including programmer and management time

Table 1. Project Size and Staff-month Effort

The results of this can be briefly summarized by Table 2. According to this, in NASA, 22% of a project's effort is during the design phase, while 49% is during coding. Integration testing takes 16% while all other activities take 12%. (Remember that requirements data are not being collected here. We are simply reporting the percentage of design, coding, and testing activities. A significant requirements activity does occur.)

	Design	Code	Int. Test.	Other
By phase	22	49	16	12
By activity	25	30	15	29

Table 2. Activities performed in each phase (by %)

However, actual activities differ somewhat from simply looking at effort spent between somewhat arbitrary calendar dates set up months in advance. By looking at all design effort across all phases of the projects, design activity is actually 25% of the total effort rather than the 22% listed above. Coding is a more reasonable 30% which means that the coding phase includes many other activities. "Other" increased from 12% to 29%, and include many time-consuming tasks that are not accounted for by the usual life cycle. (Here, Other includes acceptance testing, as well as activities that take a significant effort but are usually not separately identifiable using the standard model. These activities include meetings, training, travel, documentation, and other various activities assigned to the project.)

The situation is actually more complex than shown in Table 2. Although using Phase Date shows that total design effort differs by only 3% from the design phase effort, the distribution of design activity throughout the project is not reflected in the table. These data are presented in Table 3.

Design	Code	Int. Test	Accept. Test
50	29	20	2

Table 3. Design Activity During Life Cycle Phases (by %)

As Table 3 shows, only 50% of all design work occurs during the Design Phase and just under one third of the total design activity occurs during the coding period. Over one fifth (20%+2%) of all design occurs during testing when the system is "supposed" to be finished.

As to coding effort, Table 4 shows that while a major part, or 70% of the coding effort, does occur during the Coding Phase, almost one quarter (16%+7%) occurs during the testing periods. As expected, only a small amount of coding (7%) occurs during the design phase; however, it does indicate that some coding does begin on parts of the system while other parts are still under design.

Design	Code	Int. Test	Accept. Test
7	70	16	7

Table 4. Coding Activity during Life Cycle Phases (by %)

Similarly, Table 5 shows that significant integration testing activities (about 34%) occur before the integration testing period. Once modules have been unit tested, programmers begin to piece them together to build larger subsystems.

Design	Code	Int. Test	Accept. Test
0	34	63	3

Table 5. Integration Activity during Life Cycle Phases

3. Prototyping

As can be seen, programmers readily flow from one activity of a project to another--more like a series of rapids and not as a discrete set of waterfalls. Any model that does not reflect this cannot hope to accurately portray software development. Boehm has proposed a spiral model [BOE 86] of software development which takes some of this into account. In addition, the concept of prototyping has been proposed as an alternative concept. The remainder of this paper will address the prototyping issue.

The current model of software development is becoming even more complex. As new techniques are developed, how do they fit into the life cycle? For example, pseudocode is often written to describe a design. This pseudocode is often iterated in greater detail to evolve into the source program. However, when does pseudocode stop being design and when does it become a source program? Prototyping is another technique which doesn't fit into this model well. In a prototype, the developer builds some operational aspect of the system and then evaluates the prototype with respect to some criteria. Where does this coding and testing fit? What activity is this in the overall life cycle?

At NASA, a prototype was developed to investigate implementation strategies for a new product. In this section, the role of the prototype will be described and the resulting data collected from building the prototype will be compared with the historical life cycle data presented in the preceding section.

A prototype Flight Dynamics Analysis System (FDAS) was implemented by NASA/GSFC. Data were collected during the development of the system. For typical flight dynamics software, which NASA has considerable experience in building, prototyping would be of limited benefit due to significant knowledge of how previous systems were built. However, in this case, FDAS was to be a source code maintenance system to manage other source code libraries. It would enable NASA analysts to test new spacecraft orbit models by providing a human-engineered common interface which could be used to invoke other flight dynamics packages. Since it was unlike previous NASA projects, and since NASA personnel had limited knowledge of exactly how to build this system, FDAS was a good candidate for prototyping.

The goal of the prototype was to understand the problem domain better. As such, an early decision was made to build the system with every expectation of throwing it away. If part of the source program could be transferred to the final system, then that would be viewed as an unexpected bonus. After the prototype was built, it would be evaluated and from this experience the requirements for a production version of FDAS would be developed. Therefore, the basic idea of the prototype was to learn, and it fits into the life cycle as part of the requirements phase of Figure 2.

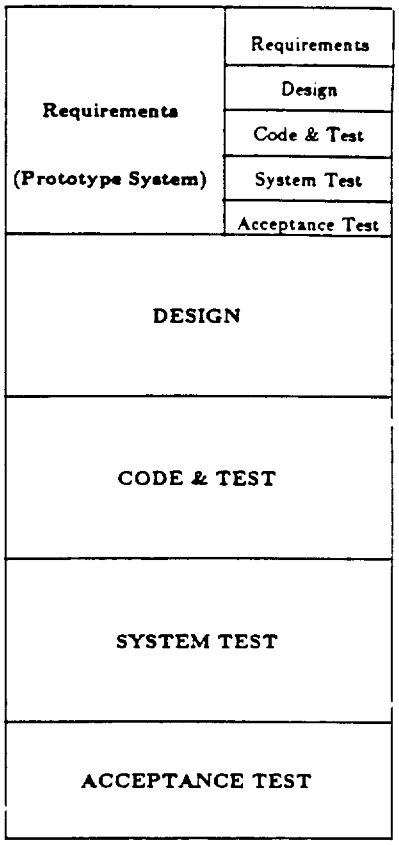

Figure 2. Prototype as part of Software Life Cycle

This definition of prototyping differs from others that view a prototype as a first release of a system. The goal was clearly to be able to understand the problem and not to generate useable source programs. In another study [BOE 84], prototyping was viewed as an iterative process converging on the final product.

We viewed the prototype as part of the requirements analysis of the problem. However, since the prototype was to execute, it itself had a full development life cycle. As Table 1 previously showed, since FDAS was almost 34,000 lines of code and took about 93 staff months to complete, it was a rather large project by itself.

FDAS was to be an interactive system. That meant that the user interface was crucial. Because of this, it was determined that the prototype should emphasize that aspect of system design.

The prototype was built in FORTRAN for a DEC VAX 11/780. In hindsight it is not clear that such an implementation was the wisest. However, at the start, the problem did not seem that complex, and personnel experience and available hardware and software lent themselves to a FORTRAN implementation. Since the goal was to give the user a taste of what services the system would provide, a screen simulation applications package (e.g., Rapid/Use [WAS 86]), a very high level simulation, or a 4th generation language might have been adequate.

The use of FORTRAN, however, did have some benefits. For one, it gave the developers experience in using FORTRAN in a type of text-processing application for which they had little previous experience. One of the reasons that the NASA group generally has high productivity is that they have had considerable experience in its application area. By building the prototype in FORTRAN, they were using Brooks' second system property where he advises "plan to throw one away" [BRO 75]. By building a first prototype in FORTRAN, mistakes would undoubtedly be made. By planning on discarding the prototype rather than patching it to correct errors, the ultimate FDAS system should be more reliable and better structured -- even if it did not turn out to be cheaper. This by itself is a valuable property, although it is not clear that it is a measurable one on most projects.

A more important aspect of a FORTRAN implementation (at least with respect to this paper) is that the FDAS prototype was a "typical" FORTRAN project. Hence its life cycle characteristics and the data that were collected could be compared with many other projects in the NASA database. This would not have been possible if some other mechanism (e.g., simulation package of some sort) were used.

In the next section, the prototype will be evaluated. However, here are some of our general conclusions. The handling of requirements differed from a production system; FDAS requirements were incomplete when design began. Unlike previous projects, they were not stated precisely because aspects of the system were still an open subject during development [ZEL 84]; even identifying the potential user community and its impact on the user interface and its effect on "assumed computer experience" was still being considered. Dates for completion of each phase were more flexible than in the historical data and milestones were less rigid than in a production development. During other phases, requirements were generally modifiable which in turn affected all activities in each phase.

More time was spent in design, than is usual for a typical project. Unlike other NASA projects, an extensive review process took place almost weekly as design decisions were made and altered. The coding and testing efforts had no formal review. Although status meetings were held almost weekly, the developers placed less emphasis on testing than with a production system; and since the prototype had a very limited lifetime, features that seemed well understood but cumbersome to implement were deleted from the requirements. According to the final report, coding took less time than in previous projects but testing did consume the same amount of effort. Very little effort was spent on acceptance testing, since the effective life of the prototype was short.

4. Evaluation of Prototype

In a manner similar to the 13 other NASA projects, the FDAS project was analyzed by phases and activities using data in the SEL database.

4.1. Phase Analysis

Data collection based on phases is shown in Table 6. The effort expended for design, coding, and testing were comparable, but notice that acceptance testing was only 1.3% of the prototype effort, but 12.7% in the historical data. With a limited lifetime, reliability was a limited feature. As long as the system worked for evaluation, it was adequate. In addition, integration testing took 10% more effort (26% compared to 16%) in the prototype. We believe this was mostly due to "schedule slippage" as the complexity of the prototype caused activities to be delayed until the end.

DEVELOPMENT EFFORT BY PHASE DATE
(13 Projects vs Prototype FDAS)

PROJECT NUMBER	DESIGN (%)	CODE (%)	INTEG. ACT.(%)	ACC.TST (%)
1	20.6	38.6	16.5	24.3
2	16.2	48.4	19.3	16.2
3	21.8	47.9	17.4	12.9
4	35.9	39.5	24.5	0.1
5	18.2	68.8	13.0	0.0
6	16.3	48.6	10.9	24.3
7	19.0	50.4	14.9	15.7
8	22.9	48.4	13.0	15.8
9	22.6	68.3	8.1	1.1
10	24.4	44.6	20.2	10.8
11	22.7	39.4	21.4	16.5
12	16.9	53.1	10.9	19.1
13	28.2	43.5	20.1	8.2
Average	22.0	49.2	16.2	12.7
FDAS	27.0	45.3	26.4	1.3

Table 6. Software Development Effort by Phase

4.2. Activity Analysis

In the previous subsection, we viewed effort by phase date. Table 7 displays the actual activities of design, coding and integration test effort independent of phase. In this case the results differ. Usually during the design phase, coding and testing activities begins on some modules, and in the code and unit test phase, additional design activity continues. Integration testing begins as soon as coding and unit testing of a component completes. Similarly, during the testing phase, any errors that were uncovered might require substantial redesign and recoding. Comparing with Table 6, we discover that most NASA developments have additional design effort later in the life cycle to raise total design effort from 22% to 25.6%. In the FDAS case, total design dropped from 27% to 25%, meaning that activities other than design occurred in the design phase. In both cases, activities other than coding occur during the coding phase since actual coding activity was only 30.5% and 17.6% respectively, as opposed to the 45+% of effort of the coding phase (Table 6).

Comparing FDAS with the 13 other developments, design effort is comparable at 25%, but the code and unit test effort and the integration test effort were different. Due to the wide variability of the "other" category of Table 7, Table 8 presents the same data as relative percent for Design, Code, and Integration testing only. This shows the differences more clearly.

No formal review was performed on the prototype during coding and unit testing. Because of the decision to delete hard-to-build but understood features that did

DEVELOPMENT EFFORT BY ACTIVITY IN ALL PHASES
(13 Projects vs Prototype FDAS)

PROJECT NUM	DESIGN ACT (%)	CODE ACT (%)	INTEG. ACT (%)	OTHER ACT (%)
1	17.4	16.4	9 9	56.3
2	30.1	39 4	20 8	9.7
3	26.3	20 3	19 3	34.2
4	27.3	28.7	6 0	38 0
5	31.0	35.5	9 4	24 1
6	14 9	21.8	24.0	39.2
7	20.2	25.9	14.3	39 6
8	11.0	13 9	9 3	65 8
9	31.3	43.5	18.9	6 4
10	38.2	37 3	6.1	18 4
11	29 3	31 0	17 2	22.5
12	23.7	46 5	24 0	5 9
13	32.6	36 3	15.6	15 6
Average	25 6	30 5	15 0	28 9
FDAS	25 0	17 6	25 1	32.3

Table 7. Software Development Effort by Activity

not effect the FDAS evaluation, coding was quite straightforward. Most of the easy coding was completed in a rather short time, and the more difficult coding aspects were simply not implemented. As Table 8 indicates, at 26% coding, FDAS had the lowest relative coding effort of any of the 14 measured projects. The next lowest was 30.8% and the average over all 13 was 42.2%. In addition, while in most projects the design and integration testing efforts were less than the coding activity, in FDAS both were almost 50% greater than for coding (about 37% for each compared to 26% for coding).

PER CENT EFFORT IN EACH PHASE
(13 Projects vs Prototype FDAS)

PROJECT NUM	DESIGN ACT(%)	CODE&UNIT ACT(%)	INTEG. ACT(%)
1	39.9	37.5	22.6
2	33.3	43.7	23.0
3	39.9	30.8	29.3
4	44.0	46.3	09.7
5	40.8	46.8	12.3
6	24.6	35.9	39.5
7	33.5	42.8	23.6
8	32.2	40.7	27.1
10	46.8	45.7	07.5
11	37.8	40.1	22.1
12	25.2	49.4	25.5
13	38.6	43.0	18.4
Average	36.2	42.2	21.6
FDAS	36.9	26.0	37.1

Table 8. Relative Activity

This apparent short circuiting of coding, however, appeared to have a detrimental effect on testing, which took a relative 37.1% of effort as opposed to 21.6% on other projects. Only one other project (6) took as much effort (39%) and from Table 1 project 6 was the most costly, where you might expect an excessive need for testing.

Based on the original productivity rate of 1.4 source lines of code (SLOC) per hour on most NASA projects [BAS 81], FDAS with a size of 33,967 SLOC had a productivity rate of 2.4 SLOC per hour. (Note: the average project size of 57,890 SLOC of Table 1 cannot simply be divided by the average effort of 13,552 hours since most NASA projects reuse some code from previous systems. Table 1 is total system size, and the productivity rate is for new lines of code.)

4.2.1. Design Effort

A true picture of development can be achieved by investigating actual activity during each phase. Although design is supposed to occur principally during the design phase, for both the 13 older projects and the FDAS prototype a comparable one half of the total design effort occurred during the design phase, and equal amounts were distributed through the rest of the life cycle (Table 9). This repeats Table 3 in more detail. Only 2% of the design of FDAS occurred during the acceptance test phase in the prototype, principally because the FDAS acceptance testing phase was so short and the few errors that were found did not get redesigned and corrected. For the historical data, the 6.4% of design occurring during acceptance testing represents errors found in testing that required source code to be redesigned.

DESIGN ACTIVITY EFFORT IN EACH PHASE
(13 Projects vs Prototype FDAS)

PROJECT NUM	DESIGN PHASE(%)	CODE PHASE(%)	INTEG. TEST(%)	ACC.TST. PHASE(%)
1	41.8	33.9	10.0	14.3
2	53.6	31.2	9.2	6.0
3	33.3	37.1	19.7	9.9
4	45.3	32.6	22.0	0.1
5	17.4	69.1	13.5	0.0
6	58.9	30.7	4.3	6.2
7	63.9	15.3	6.8	14.1
8	28.1	56.9	7.1	8.0
9	61.8	38.2	0.0	0.0
10	57.8	27.2	7.0	8.0
11	58.7	13.7	16.67	10.9
12	58.9	32.8	5.9	2.4
13	60.5	24.7	11.9	2.9
Average	49.2	34.1	10.3	6.4
FDAS	49.8	28.9	19.6	1.7

Table 9. Design Activity Effort

4.2.2. Code & Unit Test Effort

The code & unit test activities in the prototype, however, represent a departure from the older projects (Table 10). In most developments, about 7% of the coding is completed during design (although it varied from 0% to 22% in the 13 other projects). Implementation often begins as some components become completely specified. However, with FDAS, due to its greater uncertainty, no coding occurred until the development team really understood the design, i.e., until the coding phase began. For most projects, 70% of the total code and unit test effort is in the coding phase, but in the prototype almost 96% of the effort was during coding. Coding often extends through acceptance testing, but with FDAS's relatively light acceptance test, few critical errors were found so little effort was spent in recoding during test. Coding and testing need to be carried out on the full system for every change or modification of the design, but in the prototype it was not necessary to code the new design.

CODE & TEST ACTIVITY EFFORT IN EACH PHASE
(13 Projects vs Prototype FDAS)

PROJECT NUM	DESIGN PHASE(%)	CODE PHASE(%)	INTEG. TEST(%)	ACC.TST. PHASE(%)
1	1.4	78.3	11.3	9.1
2	0.0	72.8	18.7	7.5
3	22.2	56.2	11.8	9.8
4	16.4	58.6	25.1	0.1
5	21.2	68.7	10.1	0.0
6	0.5	77.3	11.3	10.9
7	1.3	73.9	15.6	9.2
8	14.7	54.7	21.0	9.7
9	5.2	91.1	3.1	0.6
10	0.0	73.0	22.5	4.5
11	2.2	70.5	20.1	7.2
12	0.3	74.8	8.3	16.6
13	4.6	63.6	26.9	4.9
Average	6.9	70.3	15.9	6.9
FDAS	0.0	95.9	4.1	0.0

Table 10. Code & Unit Test Activity Effort

4.2.3. Integration Test Effort

Integration test effort is distributed through all phases in the collected projects with more effort (43%) during the code & unit phase than in either the integration phase (26%) or the acceptance test phase (26%) (Table 11). In general, almost 50% of all integration testing occurs during design and coding phases. In FDAS, this effort was delayed with about two-thirds of all integration activities in the integration phase. This was due to delaying the integration until more pieces of the system were completed.

4.2.4. Other Activity Effort

The Other category consists of activities such as travel, completion of the data collection forms, meetings, or training. While these activities are often ignored in most life cycle studies, the costs are significant. Typically, about 29% of activities are in this category and of the 13 measured projects, "other" consumed more than one-third of the effort on 6 of them (Table 7). FDAS used a comparable 32% "other". As seen in Table 12, the prototype devoted more effort to the design phase, mainly for meeting, traveling, and training due to the extensive unknown quality of the design at the beginning

INTEGRATION ACTIVITY EFFORT IN EACH PHASE
(13 Projects vs Prototype FDAS)

PROJECT NUM	DESIGN PHASE(%)	CODE&UNIT PHASE(%)	INTEG. TEST(%)	ACC.TST. PHASE(%)
1	0.0	17.8	27.4	54.7
2	0.0	45.2	30.1	24.7
3	6.1	53.9	21.1	18.9
4	21.0	39.3	39.7	0.0
5	28.4	71.0	0.6	0.0
6	1.0	40.9	17.6	40.5
7	0.5	54.1	26.3	19.2
8	2.9	33.8	19.2	44.1
9	0.0	68.4	29.2	4.4
10	0.0	23.1	41.5	35.5
11	0.0	36.4	35.1	28.5
12	0.1	32.7	22.4	44.8
13	1.5	49.5	28.9	20.2
Average	4.7	43.4	26.1	25.8
FDAS	0.0	34.5	62.7	2.8

Table 11. Integrating Test Activity Effort

of the task. The acceptance test activity is low for the similar reason that the prototype system had few users of short duration and therefore no detailed tests. On the 13 collected projects, the Other activities are distributed more uniformly during all phases, including the acceptance test where there is a need to test before actually turning the system to the user.

OTHER ACTIVITIES EFFORT IN EACH PHASE
(13 Projects vs Prototype FDAS)

PROJECT NUM	DESIGN PHASE(%)	CODE&TST PHASE(%)	INTEG. TEST(%)	ACC.TST. PHASE(%)
1	23.3	32.2	18.1	26.5
2	0.0	9.1	26.4	64.6
3	21.7	47.8	16.8	13.7
4	46.2	30.2	23.6	0.0
5	11.0	67.7	21.3	0.0
6	18.2	44.2	9.0	28.7
7	14.4	51.6	14.5	19.5
8	26.5	47.7	11.4	14.4
9	15.9	65.5	18.7	0.0
10	12.4	30.2	35.9	21.5
11	21.4	32.2	18.9	27.6
12	47.3	46.6	4.6	1.5
13	42.5	30.0	12.7	14.9
Average	23.1	41.2	17.8	17.9
FDAS	45.1	38.8	15.7	0.3

Table 12. Other Activities Effort

5. Conclusions

In this paper we have collected data on many software projects developed at NASA/GSFC and compared them with a new prototype development. By using data from the SEL database, it appears clear that the software development process does not follow the waterfall life cycle. It also appears that the prototype development follows a similar life cycle pattern as other software projects. Although a single data point (the prototype) does not give definitive answers, it does give some trends that are of interest.

Both approaches have similar software life cycles, but the effort distributed over each phase differs. The coding in the prototype was more ad hoc, therefore testing became more involved. Integration testing was harder in the prototype because of the false assumption that reliability was not a central issue. The production developments devote more effort in coding than in testing (Table 7).

While not inexpensive, the prototype appears to be successful. Several design decisions turned out to be partially faulty when the prototype was tested. The human computer interface has been redesigned.

In fact, after completion of the prototype, several screen simulation systems were used to model a user interface, and a more hierarchical menu model was developed. Without the FDAS experience, NASA might have implemented a system where users had no real experience until the large implementation would be too far along to change adequately.

The underlying execution model of FDAS became better understood. As a source code control system, the separation of the FDAS code and the user's flight dynamics application code became clearer. Most user programs would be FORTRAN (at least initially); however, other languages (e.g., Pascal, Ada) would be used in the future, while it would not matter to the user in what language FDAS was itself written.

FDAS included a prototype preprocessor to add abstract data types to FORTRAN. This preprocessor was initially tied directly to the FDAS implementation. It is now somewhat independent to allow for other preprocessors later. The FORTRAN preprocessor, call OPAL, for Object Programming Applications Language [CSC 86], is a more rational extension of FORTRAN with data structures useful for flight dynamics applications, such as vectors, matrices, and quaternions. The decision was also made to move away from FORTRAN, and the system itself is being implemented in Ada, although it will initially process FORTRAN application code.

A new production FDAS implementation would avoid many potential pitfalls discovered via the prototype. Currently the production version of FDAS is under development, and its design has benefited greatly from the earlier development. We will have to wait for completion before fully evaluating this process. It is quite clear, however, that FDAS will be a much better product that if the prototype had not been built.

Prototyping probably increases the cost of the system, but it greatly increases its quality. It gives a flavor to the end user of what the system can do and how it can perform the task, especially in a nonfamiliar environment. It provides the developers a "second system" effect for perfecting a design.

6. Acknowledgement

This work was partially supported by grant NAG5-368 from NASA Goddard Space Flight Center to the University of Maryland. Judin Sukri provided most of the analysis of the data used in this report. We also wish to acknowledge the help of Frank McGarry of NASA/GSFC for his aid in collecting the data used here and for their interpretation.

7. References

[BAS 78]
Basili, V. R. and Zelkowitz M. V. "Analyzing Medium-Scale Software Development" 3rd International Conference on Software Engineering, Atlanta, pp. 116-223, (May 1978).

[BAS 81]
Basili, V. R. and Freburger, K., "Programming Measurement and Estimation in the Software Engineering Laboratory" *The Journal of System & Software*, Vol. 2, pp. 47-57, (1981).

[BOE 84]
Boehm B., Gray, T. and T. See Waldt, Prototyping vs. specifying: a multiproject experience, 7th International Conference on Softwrae Engineering, orlando, FLA, March 1984, pp. 473-484.

[BOE 86]
Boehm B., A spiral model of software development and enhancement, *ACM Software Engineering Notes* 11, 4, August 1986, pp. 22-42.

[BRO 75]
Brooks, F., *The Mythical Man Month* Addison Wesley, 1975.

[CSC 86]
Applications Software under the Flight dynamics analysis system (FDAS), Computer Sciences Corporation, CSC/SD-86/6024, November, 1986.

[SEL 81]
McGarry, F. E. and Page, G. and et al., "Standard Approach to Software Development", NASA Goddard Space Flight Center, Greenbelt, MD, (September 1981).

[SEL 82]
McGarry, F. E. and Church, V. and Carl, D. and et al., "Guide to Data Collection", NASA Goddard Space Flight Center, Greenbelt, Md, (August 1982).

[WAS 83]
Wasserman, A., "Software Engineering Environment", *Advances in Computer*, Vol. 22, pp. 110-159, (1983).

[WAS 86]
Wasserman A. I., P. A. Pircher and D. T. Shewmake, Building reliable interactive information systems, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, pp. 147-156 (January, 1986).

[ZEL 78]
Zelkowitz, M.V., "Perspective on Software Engineering", *ACM Computing Surveys*, Vol. 10, NO. 2, pp. 198-216, (June 1978).

[ZEL 84]
Zelkowitz, M. V. and Sukri, J., "Technique for Subjective Evaluation of Prototyping Design", *Proceeding of Ninth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, (December 1984).